Contents of lecture 2:

1. **Principles of Object Oriented Programming**
    - Object Oriented (OO) principles
    - Basics of OOP using Java

2. **OOP Java Style**
    - Classes and methods
    - Static and Final keywords
    - Objects and References within Java

Principles of Object Oriented Programming: Part 1: Abstraction and Encapsulation

Object Oriented Programming: Simple Idea

Object oriented programming (OOP) is based on a few simple ideas, all of which make good common-sense. From this point of view, OOP is nothing more than a straightforward concept (which is 100% true). However, two factors (one of which is artificial) contrive to make OOP more difficult to understand. They are:

- 'Surfeit of Zen' problem. In order to *fully* appreciate any one part, you need to understand most of the other parts.
- Use of complex, technical-sounding terminology (which does not really encapsulate the simplicity of the principles behind OO programming). See example below.

> **Example:** 'Object-Oriented Programming is characterised by inheritance and dynamic binding. C++ supports inheritance through class derivation. Dynamic binding is provided by virtual class functions. Virtual functions provide a method of encapsulating the implementation details of an inheritance hierarchy.'

Note, that the example shown above (drawn from an actual textbook) is technically accurate, however, it is also utterly useless for teaching the fundamentals of OOP. Hopefully, the coverage of OOP, and Java, within this course will try to avoid unnecessary technical baggage.

The Development of Object Oriented Languages

A fundamental and necessary part of programming is abstraction, i.e. taking an idea, concept or object and expressing it in terms of the constructs provided by the programming language (using abstraction, a person might be represented as a bundle of variables and methods, e.g. fHeight, fWeight, HairColour, eat(Food item), sleep(), run() etc.).

As programming languages have evolved they have offered more powerful and sophisticated forms of abstraction. In this section we will briefly explore how the forms of abstraction have evolved, leading to the introduction of object-oriented approaches.

Machine Language and Assembly Language

We start our brief coverage with machine language, which can be viewed as directly 'talking' to the processor in the 'language' it understands, i.e. in binary opcodes (instructions).

```
10101111
01010011
11011001
11110110
```

Understandably, programmers found machine language incredibly cumbersome to use: the strings of 1's and 0's that go together to form the op-codes of machine language are not particularly easy to remember or understand, e.g. the sequences 10111010 and 10101011 look somewhat similar to one another and their action when processed by the CPU is not readily apparent.

Assembly language was in part introduced to overcome the difficulties associated with using machine code. It was formed on the idea of assigning more meaningful labels to each machine language instruction; hence, the sequences 10111010 and 10101011 might be labeled as ADD and SUB. Assembly language can be viewed as an abstracted form of machine language as it adds an extra layer of meaning by assigning textual labels to each op-code. However, it is important to recognise that assembly language did not introduce any new functionality; it was solely designed to make the task of programming more straightforward for the programmer.

Later Imperative Languages

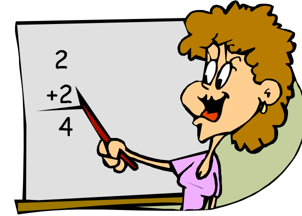
Languages which can be considered as superceding assembly language, i.e. imperative languages such as Fortran, BASIC, C, etc. all extend assembly language, not by offering any significantly new functionality, but by introducing higher level concepts (i.e. abstractions) that significantly improved the ability of the programmer to express his or her ideas in code. Some examples follow:

Data-types in assembly vs. Data-types in C

In assembly language no distinction is made between program code and data, it's all just bytes in memory. Hence, if we wish to write an assembly language program that can store several names (e.g. "Bob", "Bill" and "Sue"), it will be necessary to set aside a block of memory space in which to store the ASCII representation of each name, taking due care that the names are always correctly accessed.

Later imperative languages, such as C, enabled the programmer to declare variables, with the compiler dealing with how the data is allocated and represented in memory, and further ensuring that the data item is correctly accessed.

Adding in assembly / Adding in C



In assembly we have:

```
mov( loc1, al )
  // Move specified memory address contents into CPU register.
add( loc2, al )
  // Add specified memory address contents to CPU register.
mov( al, res )
  // Move CPU register contents into the specified memory address.
```

In C we have:

```
result = val1 + val2;
  // Add value of variable val1 to value of variable val2, storing result in specified variable.
```

Notice that the C code is considerably more compact and readily comprehended. This is in part due to it syntactical similarity to normal mathematical notation and the use of objects instead of memory offsets.

In the above examples we are not really adding anything new in terms of programming functionality, instead we are simply providing higher-level abstractions that mean the programmer can think about the problem at a higher level, without worrying about how objects, loops, etc., will be expressed in terms of machine code (i.e. the compiler does some of the work for the programmer). Apart from quickening development times and reducing the number of program bugs, such languages free up the programmer's cognitive resources to tackle larger and more difficult problems which would have been prohibitively complex to write using only assembly language.

Solving different types of problem: Non-imperative languages

Imperative languages such as assembly or C require the programmer to think about how to solve the problem in terms of the structure and capabilities of the computer. Hence, it is not sufficient for the programmer to know how to solve the problem, they must also be capable of expressing a solution in terms of basic actions that a computer can perform (e.g. adding, looping, manipulating data, etc.). Obviously for certain problems this may not always be an easy translation.

As a solution to this difficulty, a number of languages were developed that permit the programmer to more directly model the problem (i.e. at a conceptually higher level), with the compiler performing the translation into code that can be run on a CPU. Examples of such languages include LISP, APL and PROLOG, which each present different views of the world (i.e. particular approaches as to how problem solutions should be expressed). For example, LISP assumes all problems ultimately boil down to lists and list manipulation; APL assumes all problems are mathematically algorithmic in nature; and PROLOG assumes all problems can be expressed in terms of rules and decisions chains.

Each of the above non-imperative languages enables the programmer to elegantly express solutions when tackling the particular class of problem that they aim to model (the programmer does not need to worry about how the underlying implementation will be expressed in machine language). Whilst these languages often provide higher-level abstractions beyond standard imperative languages, they are limited in that design becomes cumbersome and awkward when tackling problems that fall outside of the modelled domain.

Having the entire pie, and eating it.

So far we have seen that assembly language provides a simple direct abstraction of machine code, designed to be more readily usable by the programmer. Later imperative languages, such as C, introduced higher-level notions, such as user-defined variables, various forms of looping, etc. Languages such as PROLOG deviated from standard imperative languages by offering an abstraction particularly suited to a certain class of problem. Object-oriented languages extend this idea one step further by providing higher-level abstractions, without being restricted to a certain class of problem.

This is accomplished by providing capabilities within the language (e.g. classes, etc.) whereby the programmer can define a higher-level abstraction tailored to the problem at hand. The basic idea is that the program can be adapted to the problem by defining new types of object and relationship specific to that problem (rather then being forced to use an existing, language-provided, data-type which might not be appropriate).

Obviously, the expense of this additional flexibility and modeling power is the need for the programmer to accurately model the problem within the language - a process that can be time consuming. Indeed, one of the key challenges of object-oriented programming is the creation of a one-to-one mapping between the objects and types found within the problem and those created by the programmer within the developed software.

The process of object-oriented software development can be considered as initially *modeling* the problem, and then finally *solving* the problem (both this lecture and the next lecture aim to show you how you can *model* problems using Java's object-oriented capabilities).

Object-oriented languages can be summarised by the following five key points:

1. Everything is an object – an object being an entity that can store data and can perform certain types of operation (i.e. object = black box that can store stuff and do stuff).
2. An object-oriented program involves a bunch of objects telling each other what to do by sending messages to one another (i.e. the program consists of objects working in harmony to accomplish some task). Each object can be formed as an agglomeration of other objects, whilst still being regarded as a single entity from the outside, e.g. complexity can be built up in a program while hiding it behind the simplicity of objects.
3. Every object has a type (i.e. belongs to a particular class).
4. All objects of a particular type can receive the same messages and operate in a similar manner.

Aside: Aristotle was probably the first person to carefully study the concept of type; he spoke of the 'class of fishes and the class of birds'. The idea that all objects, whilst being unique are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object oriented language Simula-67, which introduced the keyword 'class'.

Aside: Simula, as the name suggests, was created for developing simulations such as the classic 'bank teller program' where you have a bunch of tellers, customers, accounts, transactions, units of money, etc. However, whilst Simula was the first object-oriented language, it is Smalltalk that holds the honor of being the first successful and widely known OO language (in part, Java is based on Smalltalk).

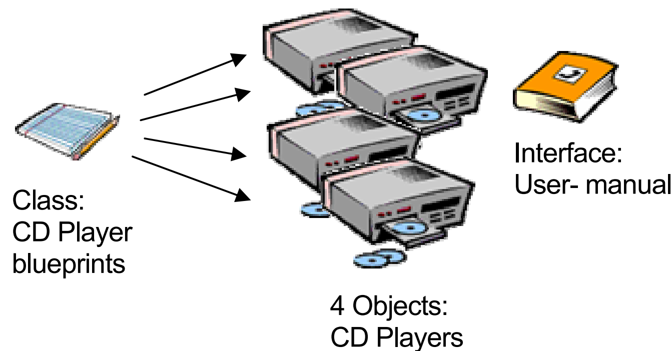
Differences between classes and objects

Objects and classes can be defined, and differentiated, as follows:

> **Class**: A template or type, describing the fields (data) and methods that are grouped together to represent something. A class is also known as a non-primitive type (more on this later).

> **Object**: A variable constructed according to a class template, able to hold values in its datafields, and able to have its methods called. There are typically many objects for any given class. We say an object is an instance of a class, or belongs to a class

Diagrammatically the relationship between objects and classes can be viewed as follows:



In order to use an object, all that is needed is knowledge of its *interface* which provides details of methods and data available to the programmer (i.e. those declared as public, etc.).

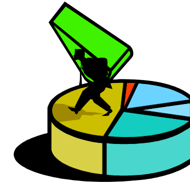
Java is a hybrid language.

Languages such as Java and C++ are considered as hybrid languages in that they permit (it would be more accurate to say 'require') multiple programming styles. For example, C++ extended C, an imperative language, by introducing object-oriented ideas. Hence, when writing a C++ program the programmer must think in terms of the classes and objects needed to solve the problem, but ultimately the classes must be expressed (written) in terms of basic CPU operations (i.e. in terms of loops, data manipulation, etc.). The same is true of Java, which drew it inspiration from a number of languages (including C++ and Smalltalk)

The first half of the pie: Abstraction and Encapsulation

Principle ingredients of OOP

There are four principle concepts upon which object oriented design and programming rest. They are: Abstraction, Polymorphism, Inheritance and Encapsulation (i.e. easily remembered as A-PIE). In what follows, each of these four concepts will be introduced and further explored.



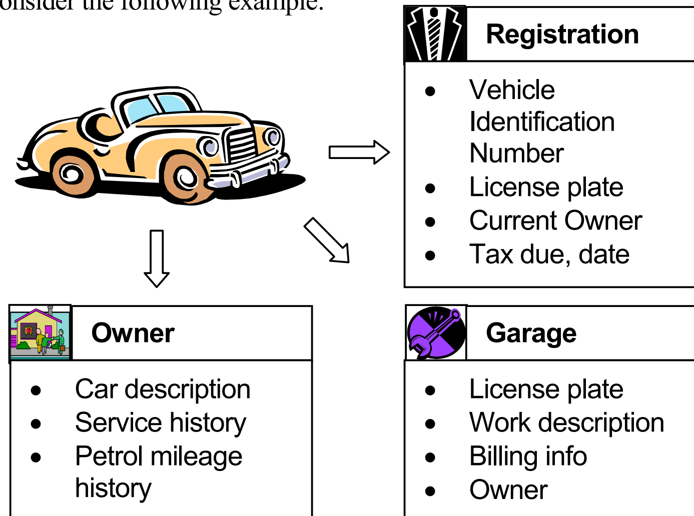
Aside: Whilst A-PIE may be easy to remember, the principles build upon one another in the following order: Abstraction, Encapsulation, Inheritance and Polymorphism.

Aside: As mentioned OOP is not a new idea; in fact it's very old (by computing standards). The first object-oriented programming language was Simula-67, produced over 30 years ago.

OOP Principle 1: Abstraction

Data Abstraction

In order to process something from the real world we have to extract the essential characteristics of that object. Consider the following example:



Data abstraction can be viewed as the process of refining away the unimportant details of an object, so that only the useful characteristics that define it remain. Evidently, this is task specific. For example, depending on how a car is viewed (e.g. in terms of something to be registered, or alternatively something to be repaired, etc.) different sets of characteristics will emerge as being important.

Formulated as such, abstraction is a very simple concept and one which is integral to all types of computer programming, object-oriented or not.

Abstraction and the programmer

The task of the programmer, given a problem, is to determine what data needs to be extracted (i.e. abstracted) in order to adequately design and ultimately code a solution. Normally, with a good problem description and/or good customer communication this process is relatively painless.

It is difficult to precisely describe how a programmer can determine which data items are important. In part it involves having a clear understanding of the problem. Equally, it involves being able to see ahead to how the problem *might* be modeled/solved. Given such understandings, it is normally possible to determine which data items will be needed to model the problem. However, please note, the ability to abstract relevant data is a skill that is largely not taught, but rather one that is refined through experience.

Failure at this stage can have two possible forms:

- Something *unnecessary* was abstracted, i.e. some data is redundant. This may or may not be a problem, but it is bad design.
- Something *needed* was *not* abstracted. This is a more serious problem, usually discovered later in the design or coding stages, and entails that the design needs to be changed to incorporate the missing data item, code changed, etc.

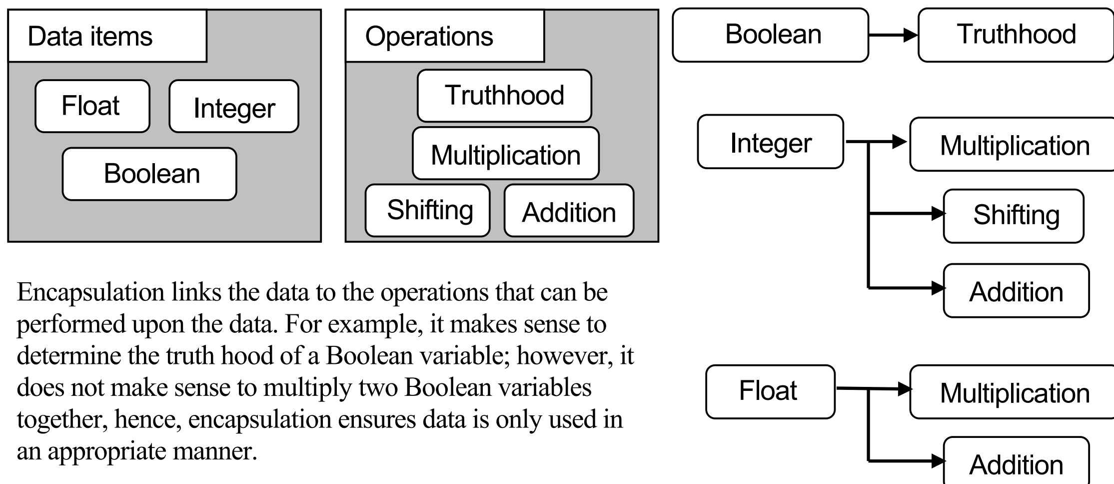
Functional Abstraction

The process of modeling functionality suffers from the same pitfalls and dangers as found when abstracting the defining characteristics, i.e. unnecessary functionality may be extracted, or alternatively, an important piece of functionality may be omitted. The programmer goes about determining which functionality is important in much the same way as they determine which data items are important.

OOP Principle 2: Encapsulation

Encapsulation is one step beyond abstraction. Whilst abstraction involves reducing a real world entity to its essential defining characteristics, encapsulation extends this idea by also modeling and *linking* the functionality of that entity.

Consider the following data items and operations:



Encapsulation links the data to the operations that can be performed upon the data. For example, it makes sense to determine the truth hood of a Boolean variable; however, it does not make sense to multiply two Boolean variables together, hence, encapsulation ensures data is only used in an appropriate manner.

Encapsulation gives classes

OOP makes use of encapsulation to enforce the integrity of a type (i.e. to make sure data is used in an appropriate manner) by preventing programmers from accessing data in a non-intended manner (e.g. asking if an Integer is true or false, etc.).

Through encapsulation, only a predetermined group of functions can access the data. The collective term for datatypes and operations (methods) bundled together with access restrictions (public/private, etc.) is a **class**.

Classes are useful for the following reasons:

- Classes can be used to model many real world entities, thereby permitting object-oriented languages to model a wide range of problems.
- Bundling data and functionality together produces self contained units which are more maintainable, reusable and deployable.
- By enforcing the idea of encapsulation upon the programmer, object oriented languages make it more difficult for the programmer to produce 'bad' code.

Object Oriented Design: An Overview

Broadly speaking, OOD (Object Oriented Design) can be broken down into the following processes:

Step 1: Identify all objects arising from the problem description.

Step 2: Initially model the problem by defining classes for each object, in terms of the data needed to represent the object and the operations that can be performed on the object (abstraction and encapsulation)

Step 3: Further model the problem by defining relationships between classes (inheritance and polymorphism).

Step 4: Devise an algorithm that instantiates a number of objects from the defined classes, and shows how these objects can interact with one another to solve the problem.

The above process is somewhat abstract, however, it simply amounts to working out what classes are needed to model the problem, what data and methods these classes should contain, how the classes should relate to one another, and finally, how instances of the classes should interact to solve the problem (i.e. you firstly model the problem, and then secondly solve the problem).

You may have noticed that there has been no coverage on how to identify which objects need to be modeled given a problem description. Unfortunately, as with determining relevant data and functionality, the ability to select appropriate objects is something that improves through the accumulation of experience (and not something that can be profitably taught).

Step 3 will be explored within the next lecture. Step 4 can only really be built up from accumulated practical experience, i.e. by actively solving problems.

Object-oriented Programming: Java Style

Class declarations within Java

Within the Java programming language, the class syntax definition is as follows:

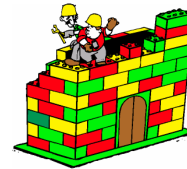
Class syntax

```
public class <classname>
{

}
```

where <classname> is the name given to the class. Java requires that each class be stored in the corresponding Java file: <classname>.java

Constructors



Each class has a constructor that is called whenever an instance of that class is created. If the class does not explicitly define a constructor then default methods are invoked (which might range from doing nothing, to calling the relevant constructor of a superclass).

Several constructors can be defined for a class, differing in terms of the parameters they accept. Constructors never return any values.

```
public class <classname>
{
        // Constructor
        public <classname>()
        { … }

        // Constructor
        public <classname>( <arglist> )
        { … }
}
```

An example showing two constructors (one the default no argument constructor, and the other accepting two integers) follows:

```
public class Counter
{
        private int iCounter, iMaximum;

        public Counter()
        { iCounter = 0; iMaximum = 100; }

        public Counter( int initCount, int initMax )
        {
                iCounter = initCount;
                setMaximum( initMax );
        }

        public void setMaximum( int iMax )
        { iMaximum = iMax; }
}
```

The procedure when an object is created follows (e.g. SomeClass myObj = new SomeClass()):

1. Sufficient memory to store the class data is allocated. This memory is then cleared, i.e. all values set to null, 0, etc. (this guarantees starting values for class data).
2. The class constructor is called. Note, the constructors of parent classes are *always* called first (either explicitly with **super(…)** or implicitly). The rest of the constructor's code is then called.

Destructors (the finalize method)

The finalize method, should one exist, is guaranteed to be called before the object is deleted.

The finalize method accepts no parameters, nor does it return any value. There can be only one finalize method for a class.

The finalize method should be used to free up any special resources which have been allocated (i.e. file handles, db locks, etc.). Thankfully, Java's built-in garage collection is very good, and the finalize method is rarely needed.

The finalize method

```
public class <classname>
{
        // Destructor
        public void finalize()
        { … }
}
```

Methods and Overloading

Apart from the constructor and finalize methods, a class consists of methods that operate upon class data. Methods can be overloaded, which is to say several different methods can have the same name, provided they differ in terms of the parameters that they accept, e.g.:

```
float getProduct( float a, float b )
{ … }

float getProduct( float a, int b )
{ … }

int getProduct( int a, int b )
{ … }
```

Note that an overloaded method cannot have several methods that share the same argument list but differ only in their return value, as there is no means of determining which method should be called, e.g.:

```
int CalculateWage( String name )
float CalculateWage( String name )
```

Data modifiers

Below, two useful data modifiers are briefly explored (data modifiers alter how items are accessed and used).

Static data, methods and classes

By declaring a class data item to be **static** it signifies that all instances of that class share the static data item in common. This is useful in a number of situations. For example, consider an Employee class, where it is important to know how many employees there are, e.g. as shown opposite:

```
public class Employee
{
        private static int iNumEmployees;

        public Person()
        { iNumEmployees++; }

        public static int numPeople()
        { return iNumEmployees; }

        // The following is a bit dangerous
        public finalize()
        { iNumEmployees--; }
}
```

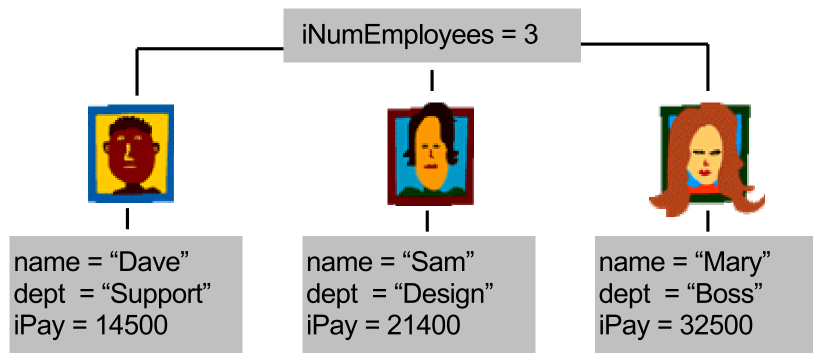
Based on the code, every time a new instance of the Employee class is instantiated the constructor increments iNumEmpolyees (a static dataitem shared across all instances of Employee).

Anytime an Employee object is deleted (by Java's garbage collection, the finalize method ensures the iNumEmployees variable is decremented). However, depending upon the CPU load, it may take a long time before the Java VM gets around to deleting the object and hence invoking the destructor.

Static data items can be visualized as follows:

```
public class  Employee
{
        public static int iNumEmployees;
        private String name;
        private String dept;
        private int iPay;

        <Methods…>
}
```



Each instance of Employee holds a certain amount of information, in particular each Employee object contains a unique name, dept and iPay variable. However, notice that the static iNumEmployees is not stored by any one Employee object, rather it is global across all instances of that class.

Note that variables declared to be static exist even if no objects of that class exist. Hence, if no Employee objects exist, then iNumEmployees = 0. Because of this a means of accessing static data when there are no instances of the class is provided. This is done by prefixing the static data item by the class name, e.g. Employee.iNumEmployees (assuming the static data has been declared to be public and hence accessible outside of the class).

A method can also be declared as static. Primarily this is of use to provide a means of accessing static data that has been declared private (class data should rarely be declared public).

```
public class Employee
{
        private static int iNumEmployees

        public static int numPeople()
        { return iNumEmployees; }
…
}
```

```
System.out.println( "Num Employees = 
Employee.numPeople()" );
```

Blocks of data and classes may also be declared to be static, however, this is of limited use.

> Aside: 'static' is not a very good name. It originated from C and referred to data that was allocated statically at compile time. Whenever you see the keyword 'static' in Java it is best to think 'once-only-per-class'

Final data, methods and classes

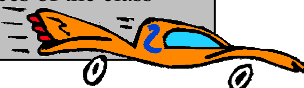
The final keyword when applied to *data* makes it read-only, i.e. it cannot be changed. Java 1.1 introduced the notion of a *blank final variable*, i.e. a variable which is blank (undefined) at initialisation, and can be assigned a value once, before then becoming read-only.

```
private final iUserInputValue;

iUserInputValue = getUserInput(); // Valid
iUserInputValue++; // Invalid now
```

Blank final variables are useful when the data value cannot be obtained before run-time, or is too complex to pre-calculate. A *method* or *class* that is defined to be final cannot be extended or modified by any sub-classes that inherit from the method or class. This is useful to ensure that the functionality of a method cannot be modified, or to ensure that a class cannot be extended via inheritance.

> Aside: final also has a number of more subtle uses. If a method or class is defined to be final then the Java Virtual Machine can execute method calls faster (this has to do with polymorphism, where Java has to check the type of each object. If declared final, Java does not need to perform the check).
>
> For example, the Java Math class is primarily declared to be final for this reason (methods within the Math class are also static, hence no instances of the class need to exist to call the method, e.g. Math.pow(), etc.).



Access Modifiers

A summary of the different access modifiers follows:

Keyword	Effect
private	Members are not accessible outside the class. Making a constructor private prevents the class from being instantiated. Making a method private means that it can only be called from within the class. private int iSum; private compute() {…}
(none) often called *package* access	Members are accessible from classes in the same package (i.e. directory) only. A class can be given either package access or public access int iSum compute()
protected	Members are accessible in the package and in subclasses of this class. Note that protected is less protected that the default package access protected int iSum; protected compute() {…}
public	Members are accessible anywhere the class is accessible. A class can be given either package or public access. public int iSum; public compute() {…}

In Java, package access broadly equates to public access between all Java files in the same directory, e.g.

```
Directory: C:/SamsProject
[3 files]
Control.java
Process.java
Printout.java

Directory: C:/JillsProject
[1 file]
Project.java
```

Assume Printout.java contains a method PrintResults() which has *package* access. If so, PrintResults can freely be called from within either the Process or Control classes. However, it cannot be called from Project as Project.java is found within a different directory.

Note, in general data should not be declared as public, instead helper read/write methods should be provided.

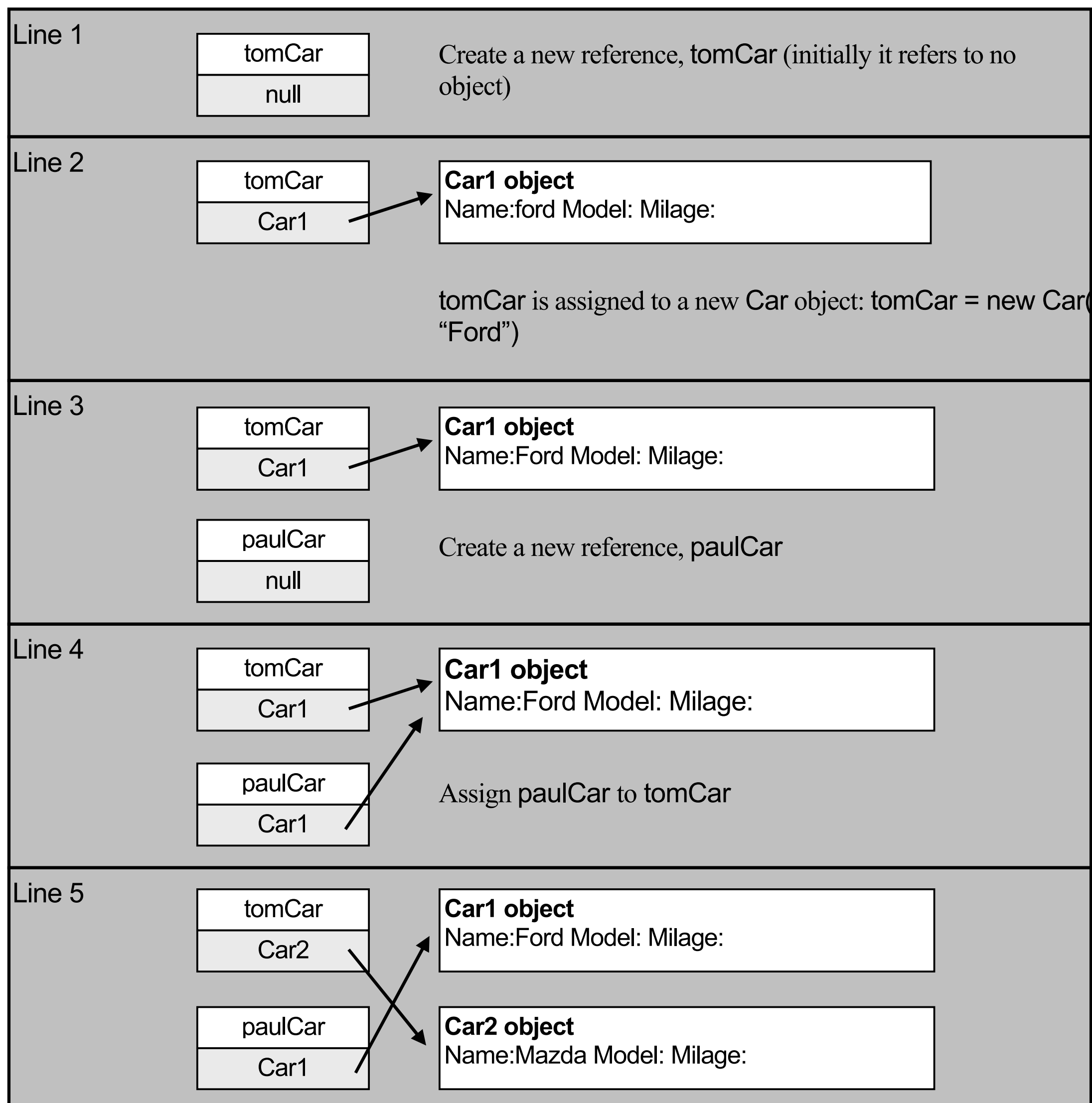
Objects within Java

Java uses *indirect addressing* when creating or manipulating objects (instantiations of a class). Otherwise stated, you can only manipulate an object through a reference variable.

What does this mean? Basically, that you never deal directly with an object, instead you always deal with an intermediary to the object. Consider the following Java statements (assuming we have a 'Car' class already defined):

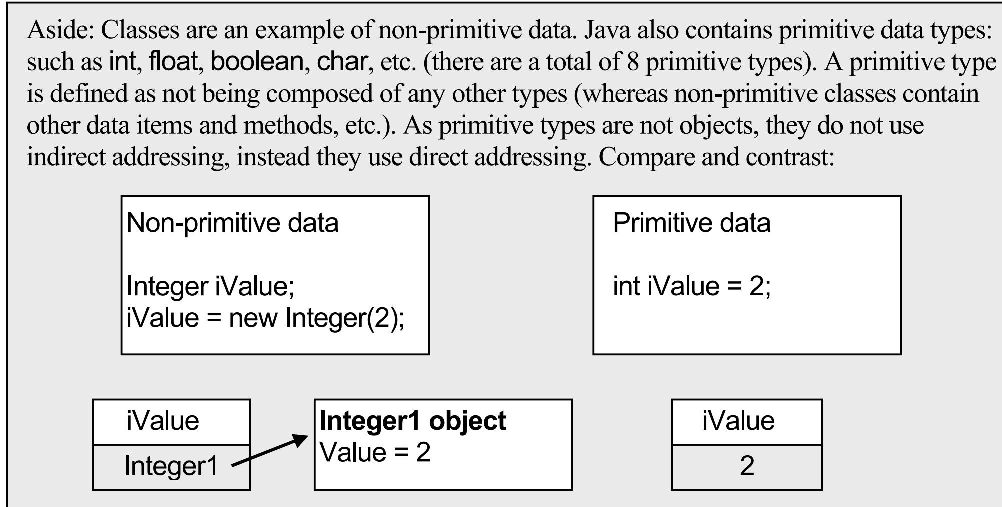
```
[1] Car tomCar;
[2] tomCar = new Car( "Ford" );
[3] Car paulCar;
[4] paulCar = tomCar;
[5] tomChar = new Car( "Mazda" );
```

The code fragment represents a fictional case where Tom originally has a Ford car, which he then shares with Paul. Finally, Tom gets a new car, a Mazda. When executed the above code will result in the following data items being created:



When a variable of any class type is declared it is simply a *reference variable* that can hold a *pointer* to an instance of the relevant class. This is a process known as *indirect addressing* (i.e. tomCar is an intermediary that holds a pointer to a Car object).

As such, this has important consequences when it comes to comparing objects, or passing objects into methods, as will be shown.

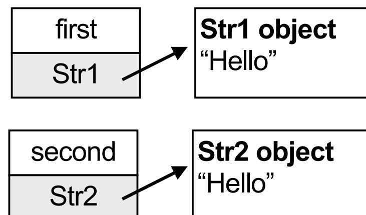


Comparing objects

Consider the following code fragment:

```
String first = new String( "Hello" );
String second = new String( "Hello"
```

Which is realised as follows:

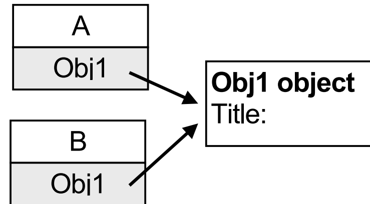


Consider the statement 'if(first == second)', will this be evaluated as true or false? In this case it will be evaluated as false, the reason being that the statement is interpreted as meaning: *Does the 'first' reference refer to the same object as the 'second' reference.* Evidently, they do not refer to the same object, hence the statement evaluates as false.

This is a common source of program errors. It is important to be aware if a comparison is to be made between references (i.e. pointers to objects) or between objects (i.e. the data internal to an object). For this example, in order to compare the object's contents we should use the String method equals, e.g. if(first.equals(second)), which compares the String's contents.

The above principle also impinges upon multiple references to the same object, e.g.:

```
Obj A = new Obj();
Obj A.setTitle( "A" );
Obj B = A;
B.setTitle( "B" );
A.getTitle() This returns 'B'
```



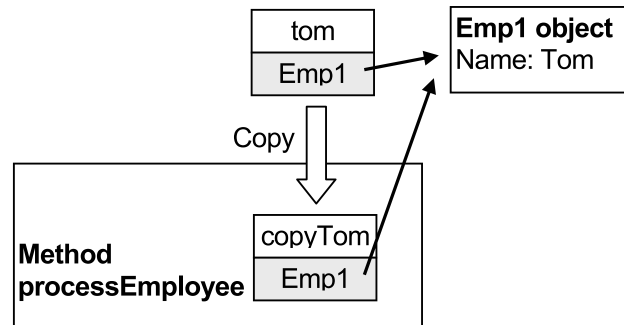
Evidently, changing an object's data through one particular reference will affect the data that is retrieved by another reference that points to the same object.

Passing references into methods

Consider the following code fragment:

```
Emplyee tom = new Employee( "Tom" );
processEmployee( tom );
```

Whenever Java calls the processEmployee method it passes a *copy* of the reference to the object, i.e.:



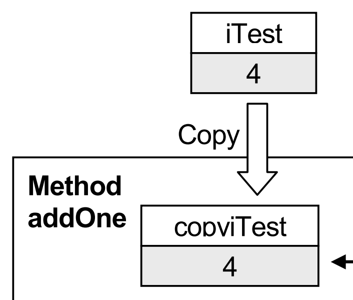
Hence, any changes made to the Employee object's data inside the processEmployee method remain after the method has returned. As such, this provides an elegant means of changing an object's data within a method. However, the programmer must be mindful not to inadvertently change an object's data from within the method.

There is one cravat to the above outlined procedure: it only applies to non-primitive data (i.e. that which is accessed through a reference). When primitive data (e.g. int, float, etc.) is passed into a method, a copy of the actual data is made, and any changes within the method are not reflected when the method returns, e.g.

```
int iTest = 4;
addOne( iTest );
System.out.println( iTest );
```

```
public addOne( int iValue )
{ iValue++ }
```

The above code will return a value of 4.



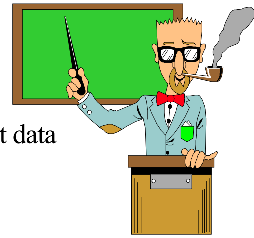
This value is changed to 5 by the method. However, once the method finishes, the value is discarded.

Practical 2

After this lecture you should explore the second practical pack which should enable you to investigate the material in this lecture.

Learning Outcomes

Once you have explored and reflected upon the material presented within this lecture and the practical pack, you should:



- Understanding the general methodology encapsulated by object-oriented programming.
- Understand the principles behind both Data Abstraction and Encapsulation and be able to apply these principles to extract relevant object data and object functionality given a straightforward problem description.
- Have knowledge of the syntax and functionality on offer within Java when constructing classes and be able to successfully employ Java to create appropriate classes given a straightforward class design.
- Understand the differences between classes, objects and references within Java (including primitive and non-primitive data items) and the consequences thereof when comparing or passing objects/references.

More comprehensive details can be found in the CSC735 Learning Outcomes document.